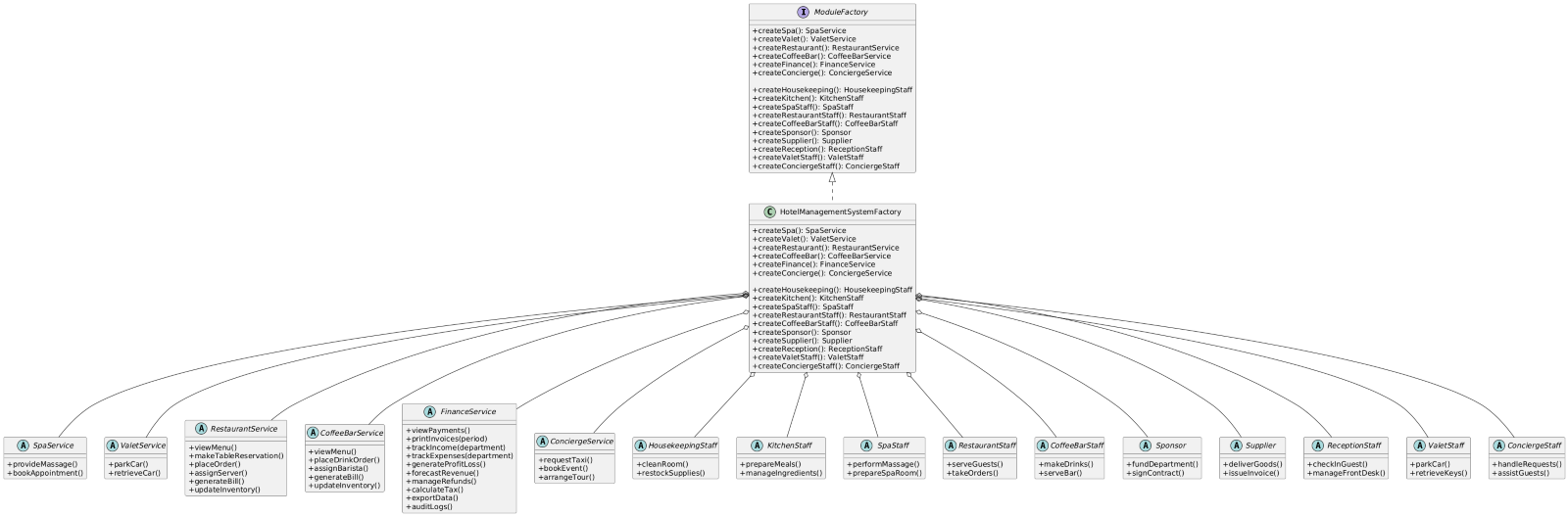
**Design Pattern**

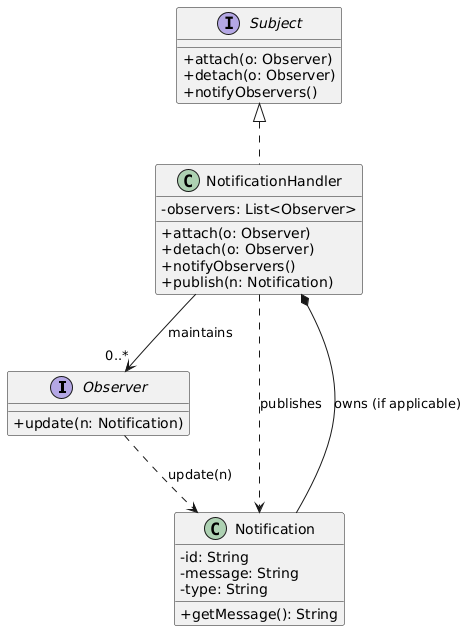
**Abstract Factory:(Arsildo Veliu-Megi Almadhi)**

The Abstract Factory pattern was used to manage the creation of all hotel services and staff types in a structured and scalable way. By defining a ModuleFactory interface with methods like createSpa() and createReceptionStaff(), we ensure a consistent method of creating related objects without exposing their specific implementations. The HotelManagementSystemFactory class implements this interface and acts as the central builder for the system. Each service or staff category is modeled as an abstract class (e.g., SpaService, HousekeepingStaff) to define required behaviors while allowing flexibility in future custom implementations. The diagram uses interface realization (dashed arrow with a triangle) to show implementation, and aggregation (white diamond) to show that the factory refers to, but does not own, the created modules. This structure keeps the system flexible and makes it easy to plug in new service types or staff roles without changing the rest of the codebase.



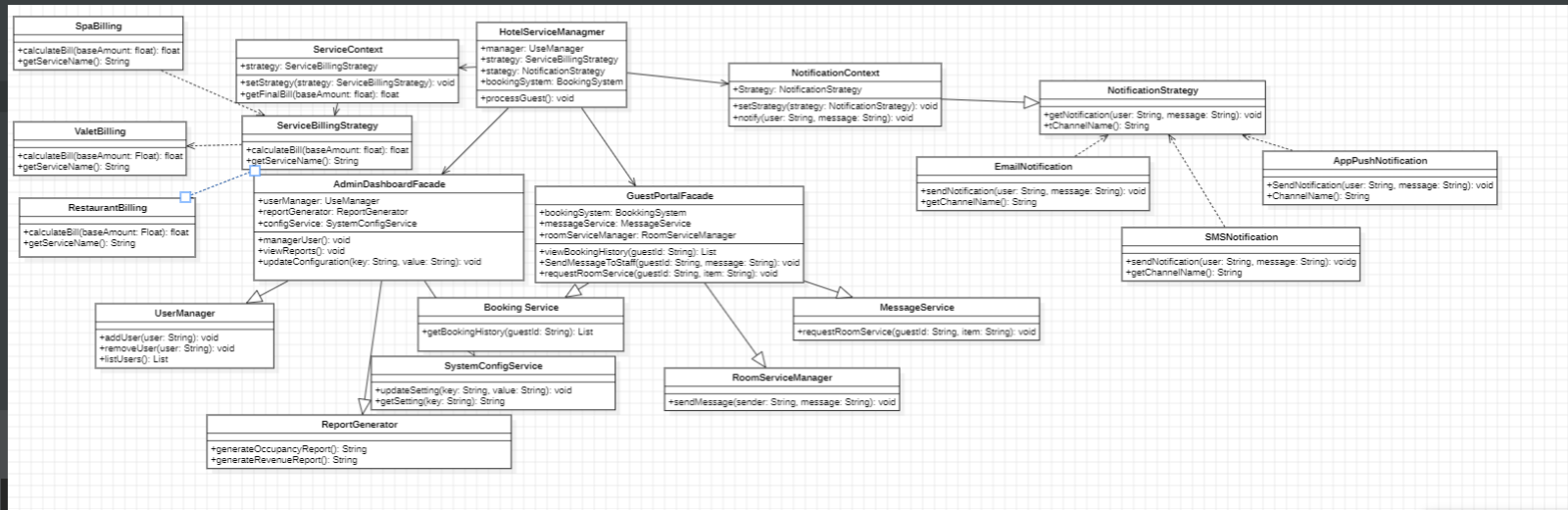
**Observer :(Arsildo Veliu,Megi Almadhi)**

The Observer pattern was used to manage notifications in the hotel system. It allows various system components, such as guests or staff, to receive real-time updates about events like bookings or room status changes. We define two key interfaces: Observer and Subject. NotificationHandler is the main class that implements Subject and manages a list of observers. It can attach or remove observers and send updates to all of them using notifyObservers(). Each observer would implement the update(notification) method to respond to messages. The Notification class carries the message data. We use interface realization to show implementation, association to indicate that NotificationHandler holds multiple observers, and dependency arrows to show that both NotificationHandler and Observer rely on the Notification object. This setup creates a flexible, decoupled system where observers can be added or removed without changing the publisher’s logic.



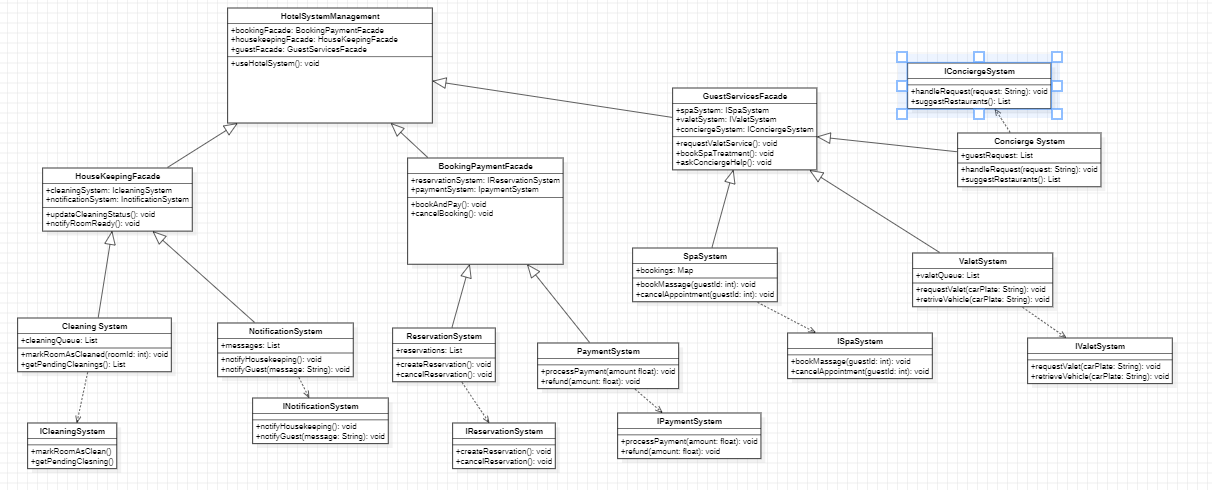
**Strategy :(Brikena Papadhopuli,Ester Pashtranjaku)**

In this hotel management system, the Strategy pattern is applied to support dynamic and interchangeable behaviors such as billing, payment, and room assignment. Each behavior is represented by an interface (e.g., BillingStrategy, PaymentStrategy), and multiple concrete classes implement these interfaces to provide specific algorithms (e.g., PremiumBilling, CashPayment, VIPRoomAssignment). These strategy objects are used through context classes that hold them via aggregation, allowing the system to switch behaviors at runtime without modifying the context logic. This approach follows object-oriented principles such as the Open/Closed Principle and improves flexibility, scalability, and testability of the system.



**Facade :(Artemisa Hasalami)**

The Facade pattern is used to simplify and organize access to complex hotel subsystems such as reservation, payment, cleaning, valet, spa, and concierge services. Instead of exposing these subsystems directly to higher-level classes, the system introduces facades like BookingPaymentFacade, HousekeepingFacade, and GuestServicesFacade. Each facade composes its related subsystems using composition, meaning the subsystems are fully managed and encapsulated within the facade. Additionally, interfaces are defined for both subsystems and facades to support abstraction and dependency injection, allowing easy testing and future extensions. This design reduces coupling between components, promotes clean architecture, and improves maintainability by hiding internal complexity.



**Singleton :(Erta Llenga,Esta Cekrezi)**

The Singleton Pattern is used in three critical classes: SharedConfiguration, StatsDashboard, and Logging. These components need to exist as single instances to maintain consistent settings, centralized performance data, and unified system logs. For example, SharedConfiguration ensures all modules operate with the same settings, while StatsDashboard provides real-time performance metrics from a single source. Logging collects all system events, essential for debugging and audits.Each singleton class implements a specific interface—IConfigurable, IReportable, or ILoggable. This allows for flexibility and clear separation of responsibilities. It also makes future changes or extensions easier, as interfaces define expected behavior without tying down the implementation.The Interface Segregation Principle ensures each class implements only the interfaces relevant to its purpose. This avoids unnecessary complexity and keeps the system modular and clean. For instance, a class focused on reporting doesn’t need to implement configuration methods.Inheritance is mainly used to connect interfaces with the classes that implement them. This allows for general behavior to be defined through interfaces and customized by specific classes like SharedConfiguration or Statistics.

